Integrity, Hashes, & "Random" Numbers

I DON'T ALWAYS ENCRYPT

BUT WHEN I DO, IT'S AES-CFB + HMAC_SHA256
Mallory the Manipulator

- Mallory is an active attacker
  - Can introduce new messages (ciphertext)
  - Can “replay” previous ciphertexts
  - Can cause messages to be reordered or discarded

- A “Man in the Middle” (MITM) attacker
  - Can be much more powerful than just eavesdropping
Encryption Does Not Provide Integrity

• Simple example: Consider a block cipher in CTR mode...
• Suppose Mallory knows that Alice sends to Bob “Pay Mal $0100”. Mallory intercepts corresponding C
  • M = “Pay Mal $0100”. C = “r4ZC#jj8qThMK”
  • $M_{10..13} = “0100”$. $C_{10..13} = “ThMK”$
• Mallory wants to replace some bits of C...
Encryption Does Not Provide Integrity

• Mallory computes
  • “0100” ⊕ C_{10..13}
    • Tells Mallory that section of the counter XOR:
      Remember that CTR mode computes E_k(IV||CTR) and XORs it with the corresponding part of the message
    • C'_{10..13} = "9999" ⊕ “0100” ⊕ C_{10..13}
    • Mallory now forwards to Bob a full C' = C_{0..9}||C'_{10..13}||C_{14...}
  • Bob will decrypt the message as "Pay Mal $9999"...
    • For a CTR mode cipher, Mallory can in general replace any known message M with a message M' of equal length!
Integrity and Authentication

- Integrity: Bob can confirm that what he’s received is exactly the message \( M \) that was originally sent
- Authentication: Bob can confirm that what he’s received was indeed generated by Alice
- Reminder: for either, confidentiality may-or-may-not matter
  - E.g. conf. not needed when Mozilla distributes a new Firefox binary
- Approach using symmetric-key cryptography:
  - Integrity via MACs (which use a shared secret key \( K \))
  - Authentication arises due to confidence that only Alice & Bob have \( K \)
- Approach using public-key cryptography (later on):
  - “Digital signatures” provide both integrity & authentication together
- Key building block: cryptographically strong hash functions
Hash Functions

• **Properties**
  • Variable input size
  • Fixed output size (e.g., 256 bits)
  • Efficient to compute
  • Pseudo-random (mixes up input extremely well)

• Provides a “fingerprint” of a document
  • E.g. “shasum -a 256 <exams/mt1-solutions.pdf” prints
    0843b3802601c848f73ccb5013afa2d5c4d424a6ef477890ebf8db9bc4f7d13d
Cryptographically Strong Hash Functions

- A collision occurs if \( x \neq y \) but \( \text{Hash}(x) = \text{Hash}(y) \)
- Since input size > output size, collisions do happen
- A cryptographically strong \( \text{Hash}(x) \) provides three properties:
  - One-way: \( h = \text{Hash}(x) \) easy to compute, but not to invert.
  - Intractable to find any \( x' \) s.t. \( \text{Hash}(x') = h \), for a given \( h \)
  - Also termed “preimage resistant”

\[ H(🐮) = \]
Cryptographically Strong Hash Functions

- The other two properties of a cryptographically strong Hash(x):
  - Second preimage resistant: given x, intractable to find x' s.t. \( \text{Hash}(x) = \text{Hash}(x') \)
  - Collision resistant: intractable to find any x, y s.t. \( \text{Hash}(x) = \text{Hash}(y) \)
  - Collision resistant \(\implies\) Second preimage resistant
  - We consider them separately because given Hash might differ in how well it resists each
  - Also, the Birthday Paradox means that for n-bit Hash, finding x-y pair takes only \( \approx 2^{n/2} \) pairs
    - Vs. potentially \( 2^n \) tries for x': \( \text{Hash}(x) = \text{Hash}(x') \) for given x
Cryptographically Strong Hash Functions, con’t

• Some contemporary hash functions
  • MD5: 128 bits
    • broken – lack of collision resistance
    • Collisions for the heck of it: https://shells.aachen.ccc.de/~spq/md5.gif
      An MD5 "hash quine": an animated GIF that shows its own hash
  • SHA-1: 160 bits broken (as of spring 2017, but was known to be weak yet still used…)
  • SHA-256/SHA-384/SHA-512: 256, 384, 512 bits in the SHA-2 family, at least not currently broken
  • SHA-3: New standard! Yayyy!!!! (Based on Keccak, again 256b, 384b, and 512b options)

• Provide a handy way to unambiguously refer to large documents
  • If hash can be securely communicated, provides integrity
    • E.g. Mozilla securely publishes SHA-256(new FF binary)
    • Anyone who fetches binary can use “cat binary | shasum -a 256” to confirm it’s the right one, untampered

• Not enough by themselves for integrity, since functions are completely known – Mallory can just compute revised hash value to go with altered message
SHA-256...

- SHA-256/SHA-384 are two parameters for the SHA-2 hash algorithm, returning 256b or 384b hashes
  - Works on blocks with a truncation routine to make it act on sequences of arbitrary length
- Is vulnerable to a length-extension attack: $s$ is secret
  - Mallory knows $\text{len}(s)$, $H(s)$
  - Mallory can use this to calculate $H(s||M)$ for an $M$ of Mallory's construction
    - Works because all the internal state at the point of calculating $H(s||...)$ is derivable from $H(s)$ and $\text{len}(s)$
- New SHA-3 standard (Keccak) does not have this property
Stupid Hash Tricks: Sample A File...

- BlackHat Dude claims to have 150M records stolen from Equifax...
  - How can I as a reporter verify this?
- Idea: If I can have the hacker select 10 *random* lines...
  - And in selecting them also say something about the size of the file...
  - Voila! Verify those lines and I now know he's not full of BS
- Can I use hashing to write a small script which the BlackHat Dude can run?
  - Where I can easily verify that the 10 lines were sampled at random, and can't be faked?
Sample a File

#!/usr/bin/env python
import hashlib, sys
hashes = {}

for line in sys.stdin:
    line = line.strip()
    for x in range(10):
        tmp = "%s-%i" % (line, x)
        hashval = hashlib.sha256(tmp)
        h = hashval.digest()
        if x not in hashes or hashes[x][0] > h:
            hashes[x] = (h, hashval, tmp)

for x in range(10):
    h, hashval, val = hashes[x]
    print "%s="%s" " % (hashval.hexdigest(), val)
Why does this work?

• For each \( x \) in range 0-9...
  • Calculates \( H(\text{line}||x) \)
  • Stores the lowest hash matching so far

• Since the hash appears random...
  • Each iteration is an independent sample from the file
  • The expected value of \( H(\text{line}||x) \) is a function of the size of the file:
    More lines, and the value is smaller

• To fake it...
  • Would need to generate fake lines, \textit{and see if the hash is suitably low}
  • Yet would need to make sure these fake lines semantically match!
    • Thus you can't just go "John Q Fake", "John Q Fakke", "Fake, John Q", etc...
Message Authentication Codes (MACs)

- Symmetric-key approach for integrity
  - Uses a shared (secret) key $K$

- Goal: when Bob receives a message, can confidently determine it hasn’t been altered
  - In addition, whomever sent it must have possessed $K$
    ($\Rightarrow$ message authentication, sorta...)

- Conceptual approach:
  - Alice sends $\{M, T\}$ to Bob, with tag $T = \text{MAC}(K, M)$
    - Note, $M$ could instead be $C = E_{k'}(M)$, but not required
  - When Bob receives $\{M', T'\}$, Bob checks whether $T' = \text{MAC}(K, M')$
    - If so, Bob concludes message untampered, came from Alice
    - If not, Bob discards message as tampered/corrupted
Requirements for Secure MAC Functions

- Suppose MITM attacker Mallory intercepts Alice’s \( \{M, T\} \) transmission …
  - … and wants to replace \( M \) with altered \( M^* \)
  - … but doesn’t know shared secret key \( K \)

- We have secure integrity if MAC function \( T = MAC(M, K) \) has two properties:
  - Mallory can’t compute \( T^* = MAC(M^*, K) \)
    - Otherwise, could send Bob \( \{M^*, T^*\} \) and fool him
  - Mallory can’t find \( M^{**} \) such that \( MAC(M^{**}, K) = T \)
    - Otherwise, could send Bob \( \{M^{**}, T\} \) and fool him

- These need to hold even if Mallory can observe many \( \{M_i, T_i\} \) pairs, including for \( M_i \)'s she chose
MAC then Encrypt or Encrypt then MAC

- You should **never** use the same key for the MAC and the Encryption
  - Some MACs will break completely if you reuse the key
  - Even if it is *probably* safe (eg, AES for encryption, HMAC for MAC) its still a bad idea
- **MAC then Encrypt:**
  - Compute $T = \text{MAC}(M, K_{\text{mac}})$, send $C = E(M||T, K_{\text{encrypt}})$
- **Encrypt then MAC:**
  - Compute $C = E(M, K_{\text{encrypt}})$, $T = \text{MAC}(M, K_{\text{mac}})$, send $C||T$
  - Theoretically they are the same, but...
  - Once again, its time for ...
HTTPS Authentication in Practice

- When you log into a web site, it sets a "cookie" in your browser
  - All subsequent requests include this cookie so the web server knows who you are
- If an attacker can get your cookie...
  - They can impersonate you on the "Secure" site
- And the attacker can create multiple tries
  - On a WiFi network, inject a bit of JavaScript that repeatedly connects to the site
  - While as a man-in-the-middle to manipulate connections

![Image: F-U!! THIS IS CRYPTO!!!]
The TLS 1.0 "Lucky13" Attack: "F-U, This is Cryptography"

- HTTPS/TLS uses MAC then Encrypt
  - With CBC encryption
- The Lucky13 attack changes the cipher text in an attempt to discover the state of a byte
  - But can't predict the MAC
  - The TLS connection retries after each failure so the attacker can try multiple times
    - Goal is to determine the status each byte in the authentication cookie which is in a known position
- It detects the **timing** of the error response
  - Which is different if the guess is right or wrong
    - Even though the underlying algorithm was "proved" secure!
- So always do Encrypt then MAC since, once again, it is more mistake tolerant
AES-EMAC: Building a MAC out of a secure block cipher

Takes 256-bit key $K$, split into two 128-bit AES keys, $K_1$ and $K_2$

Provably secure if AES is secure:

IS reversible for a single block only
The best MAC construction: HMAC

- Idea is to turn a hash function into a MAC
  - Since hash functions are often much faster than encryption
  - While still maintaining the properties of being a cryptographic hash
- Reduce/expand the key to a single hash block
- XOR the key with the i_pad
  - 0x363636... (one hash block long)
- Hash ((K ⊕ i_pad) || message)
- XOR the key with the o_pad
  - 0x5c5c5c...
- Hash ((K ⊕ o_pad) || first hash)

```javascript
function hmac (key, message) {
  if (length(key) > blocksize) {
    key = hash(key)
  }
  while (length(key) < blocksize) {
    key = key || 0x00
  }
  o_key_pad = 0x5c5c... ⊕ key
  i_key_pad = 0x3636... ⊕ key
  return hash(o_key_pad ||
              hash(i_key_pad || message))
}
```
Why This Structure?

• i_pad and o_pad are slightly arbitrary
  • But it is necessary for security for the two values to be different
  • So for paranoia chose very different bit patterns

• Second hash prevents appending data
  • Otherwise attacker could add more to the message and the HMAC and it would still be a valid HMAC for the key
  • Wouldn’t be a problem with the key at the end but at the start makes it easier to capture intermediate HMACs

• Is a Pseudo Random Function if the underlying hash is a PRF
  • AKA if you can break this, you can break the hash!

```javascript
function hmac (key, message) {
  if (length(key) > blocksize) {
    key = hash(key)
  }
  while (length(key) < blocksize) {
    key = key || 0x00
  }
  o_key_pad = 0x5c5c... ⊕ key
  i_key_pad = 0x3636... ⊕ key
  return hash(o_key_pad ||
              hash(i_key_pad || message))
}
```
Great Properties of HMAC...

- It is still a hash function!
  - So all the good things of a cryptographic hash:
    An attacker or even the recipient shouldn't be able to calculate $M$ given $\text{HMAC}(M,K)$
  - An attacker who doesn't know $K$ can't even verify if $\text{HMAC}(M,K) == M$
    - Very different from the hash alone, and potentially very useful:
      Attacker can't even brute force try to find $M$ based on $\text{HMAC}(M,K)$!
- It's probably safe if you screw up and use the same key for both MAC and Encrypt
  - Since it is a different algorithm than the encryption function...
  - *But you shouldn't do this anyway!*
Considerations when using MACs

• Along with messages, can use for data at rest
  • E.g. laptop left in hotel, providing you don’t store the key on the laptop
  • Can build an efficient data structure for this that doesn’t require re-MAC’ing over entire disk image when just a few files change

• MACs in general provide no promise not to leak info about message
  • Though the ones we’ve seen don’t if the key is secret
  • Compute MAC on ciphertext if this matters
  • Or just use HMAC, which **does** promise not to leak info if the underlying hash function doesn’t

• **NEVER** use the same key for MAC and Encryption...
  • Known "FU-this-is-crypto" scenarios reusing an encryption key for MAC in some algorithms when its the same underlying block cipher for both
Passwords

• The password problem:
  • User Alice authenticates herself with a password $P$

• How does the site verify later that Alice knows $P$?

• Classic:
  • Just store $\{\text{Alice, } P\}$ in a file...

• But what happens when the site is hacked?
  • The attacker now knows Alice's password!

• Enter "Password Hashing"
Password Hashing

- Instead of storing \{Alice, \(P\}\)...
  - Store \{Alice, \(H(P)\}\}
- To verify Alice, when she presents \(P\)
  - Compute \(H(P)\) and compare it with the stored value
- Problem: Brute Force tables...
  - Most people chose bad passwords...
    And these passwords are known
  - Bad guy has a huge file...
    - \(H(P1), P1\)
    - \(H(P2), P2\)
    - \(H(P3), P3\)...
  - Ways to make this more efficient ("Rainbow Tables")
A Sprinkle of Salt...

- Instead of storing \{Alice, H(P)\}, also have a user-specific string, the "Salt"
- Now store \{Alice, Salt, H(P||Salt)\}
- The salt ideally should be both long and random, but it isn't considered "secret"
- As long as the salt is unique...
  - An attacker who captures the password file has to *brute force* Alice's password on its own
  - It's still an "off-line attack" (Attacker can do all the computation he wants) but...
    - At least the attacker can't *precompute* possible solutions
Slower Hashes...

- Most cryptographic hashes are designed to be **fast**
  - After all, that is the point: they should not only turn $H(🐮)$ to hamburger...
    - they need to do it quickly
  - But for password hashes, we **want** it to be slow!
    - Its OK if it takes a good fraction of a second to **check** a password
      - Since you only need to do it once for each legitimate usage of that password
      - But the attacker needs to do it for each password he wants to try
  - Slower hashes don't change the **asymptotic difficulty** of password cracking but can have huge practical impact
    - Slow rate by a factor of 10,000 or more!
PBKDF2

- "Password Based Key Derivation Function 2"
  - Designed to produce a long "random" bitstream derived from the password
  - Used for both a password hash and to generate keys derived from a user's password

PKBDF(PRF, P, S, c, len):
  - PRF == Pseudo Random Function (e.g. HMAC-SHA256)
  - P == Password
  - S == Salt
  - c == Iteration count
  - len == Number of bits/bytes requested
  - DK == Derived Key

```python
PKBDF(PRF, P, S, c, len):
    DK = ""
    for i = 1,range(len/blocksize)+1:
        DK = DK || F(PRF, P, S, c, i)
    return DK[0:len]

F(PRF, P, S, c, i):
    UR = U = PRF(P, S||INT_32(i))
    for j = 2; j <= c; ++j {
        U = PRF(P, U)
        UR = UR ^ U
    }
    return UR
```
Comments on PBKDF2

- Allows you to get effectively an arbitrary long string from a password
  - Assuming the user's password is strong/high entropy
- Very good for getting a bunch of symmetric keys from a single password
  - You can also use this to seed a pRNG for generating a "random" public/private key pair
- Designed to be slow in computation...
  - But it does not require a lot of memory:
    Other functions are also expensive in memory as well, e.g. scrypt.
Passwords...

- If an attacker can do an **offline** attack, your password must be **really good**
  - Attacker simply tries a huge number of passwords in parallel using a GPU-based computer
  - So you need a **high entropy** password:
    - Even xkcd-style is only 10b/word, so need a 7 or more **random word** passphrase to resist a determined attacker

- Life is far better is if the attacker can only do **online** attacks:
  - Query the device and see if it works
  - Now limited to a few tries per second and **no parallelism!**
... and iPhones

- Apple's security philosophy:
  - In your hands, the phone should be everything
  - In anybody else's, it should (ideally) be an inert "brick"

- Apple uses a small co-processor in the phone to handle the cryptography
  - The "Secure Enclave"

- The rest of the phone is untrusted
  - Notably the memory: \textit{All} data must be encrypted:
    The CPU requests that the Secure Enclave unencrypt data and some data (e.g., your credit card for ApplePay) is only readable by the Secure Enclave

- They also have an ability to effectively erase a small piece of memory
  - "Effaceable Storage": this takes a good amount of EE trickery
Crypto and the iPhone Filesystem

• A lot of keys encrypted by keys...
  • But there is a random master key, $k_{phone}$, that is the root of all the other keys

• Need to store $k_{phone}$ encrypted by the user's password in the flash memory
  • $\text{PBKDF2}(P,...) = k_{user}$

• But how to prevent an off-line brute-force attack?
  • Also have a 256b random secret burned into the Secure Enclave
    • Need to take apart the chip to get this!

• Now the user key is not just a function of P, but P||secret
  • Without the secret, can not do an offline attack

• All online attacks have to go through the secure enclave
  • After 5 tries, starts to slow down
  • After 10 tries, can (optionally) nuke $k_{phone}$!
    • Erase just that part of memory -> effectively erases the entire phone!
Backups...

• Of course there is a necessary weakness:
  • Backing up the phone copies all the data off in a form not encrypted using the in-chip secret
    • After all, you need to be able to recover it onto a new phone!

• So someone who can get your phone...
  And can somehow managed to have it unlocked
  • Thief, abusive boyfriend, cop...
    • Hold it up to your face (iPhone X) or Fingerprint (5s or beyond)
    • And then sync it with a new computer

• Change of policy for iOS-11:
  • Now you also need to put in the passcode to trust a new computer:
    Can't create a backup without knowing the passcode
But A Lot More Uses for Random Numbers...

- The key foundation for all modern cryptographic systems is often not encryption but these "random" numbers!
- So many times you need to get something random:
  - A random cryptographic key
  - A random initialization vector
  - A "nonce" (use-once item)
  - A unique identifier
  - Stream Ciphers
- If an attacker can **predict** a random number things can catastrophically fail
Breaking Slot Machines

• Some casinos experienced unusual bad "luck"
  • The suspicious players would wait and then all of a sudden try to play

• The slot machines have **predictable** pRNG
  • Which was based on the current time & a seed

• So play a little...
  • With a cellphone watching
  • And now you know when to press "spin" to be more likely to win

• Oh, and this **never** effected Vegas!
  • *Evaluation standards* for Nevada slot machines specifically designed to address this sort of issue
Breaking Bitcoin Wallets

- blockchain.info supports "web wallets"
- Javascript that protects your Bitcoin
- The private key for Bitcoin needs to be random
- Because otherwise an attacker can spend the money
- An "Improvement" [sic] to the RNG reduced the entropy (the actual randomness)
- Any wallet created with this improvement was brute-forceable and could be stolen
TRUE Random Numbers

• True random numbers generally require a physical process
• Common circuit is an unusable ring oscillator built into the CPU
  • It is then sampled at a low rate to generate true random bits which are then fed into a pRNG on the CPU
• Other common sources are human activity measured at very fine time scales
  • Keystroke timing, mouse movements, etc
    • "Wiggle the mouse to generate entropy for a key"
  • Network/disk activity which is often human driven
• More exotic ones are possible:
  • Cloudflare has a wall of lava lamps that are recorded by a HD video camera which views the lamps through a rotating prism: It is just one source of the randomness
Combining Entropy

- The general procedure is to combine various sources of entropy
- The goal is to be able to take multiple crappy sources of entropy
  - Measured in how many bits:
    A single flip of a coin is 1 bit of entropy
  - And combine into a value where the entropy is the minimum of the sum of all entropy sources (maxed out by the # of bits in the hash function itself)
  - \(N-1\) bad sources and \(1\) good source -> good pRNG state
Pseudo Random Number Generators (aka Deterministic Random Bit Generators)

• Unfortunately one needs a **lot** of random numbers in cryptography
  • More than one can generally get by just using the physical entropy source

• Enter the pRNG or DRBG
  • If one knows the state it is entirely predictable
  • If one doesn't know the state it should be indistinguishable from a random string

• Three operations
  • Instantiate: (aka Seed) Set the internal state based on the real entropy sources
  • Reseed: Update the internal state based on both the previous state and *additional entropy*
    • The big different from a simple stream cipher
  • Generate: Generate a series of random bits based on the internal state
    • Generate can also optionally add in additional entropy

• `instantiate(entropy)`
• `reseed(entropy)`
• `generate(bits, {optional entropy})`
Properties for the pRNG

• Can a pRNG be truly random?
  • No. For seed length $s$, it can only generate at most $2^s$ distinct possible sequences.

• A cryptographically strong pRNG “looks” truly random to an attacker
  • Attacker *cannot distinguish* it from a random sequence:
    If the attacker can tell a sufficiently long bitstream was generated by the pRNG instead of a truly random source it isn't a good pRNG
Prediction and Rollback Resistance

- A pRNG should be predictable only if you know the internal state
  - It is this predictability which is why it’s called "pseudo"
- If the attacker does not know the internal state
  - The attacker should not be able to distinguish a truly random string from one generated by the pRNG
- It should also be rollback-resistant
  - Even if the attacker finds out the state at time T, they should not be able to determine what the state was at T-1
  - More precisely, if presented with two random strings, one truly random and one generated by the pRNG at time T-1, the attacker should not be able to distinguish between the two
Why "Rollback Resistance" is Essential

- Assume attacker, at time $T$, is able to obtain all the internal state of the pRNG
- How? E.g. the pRNG screwed up and instead of an IV, released the internal state, or the pRNG is bad...
- Attacker observes how the pRNG was used
  - $T_{-1} = \text{Session key}$
  - $T_0 = \text{Nonce}$
- Now if the pRNG doesn't resist rollback, and the attacker gets the state at $T_0$, attacker can know the session key! And we are back to...
More on Seeding and Reseeding

• Seeding should take all the different physical entropy sources available
  • If one source has 0 entropy, it *must not* reduce the entropy of the seed
  • We can shove a whole bunch of low-entropy sources together and create a high-entropy seed

• Reseeding *adds* in even more entropy
  • $F(\text{internal\_state}, \text{new material})$
  • Again, even if reseeding with 0 entropy, it *must not* reduce the entropy of the seed
Probably the best pRNG/DRBG: HMAC_DRBG

- Generally believed to be the best
  - *Accept no substitutes!*
- Two internal state registers, $V$ and $K$
  - Each the same size as the hash function's output
- $V$ is used as (part of) the data input into HMAC, while $K$ is the key
- If you can break this pRNG you can **either break the underlying hash function** or **break a significant assumption about how HMAC works**
  - Yes, security proofs sometimes are a very good thing and actually do work
HMAC_DRBG Generate

- The basic generation function
- Remarks:
  - It requires one HMAC call per blocksize-bits of state
  - Then two more HMAC calls to update the internal state
- Prediction resistance:
  - If you can distinguish new $K$ from random when you don't know old $K$:
    You've distinguished HMAC from a random function! Which means you've either broken the hash or the HMAC construction
- Rollback resistance:
  - If you can learn old $K$ from new $K$ and $V$:
    You've reversed the hash function!

```python
function hmac_drbg_generate (state, n) {
    tmp = ""
    while(len(tmp) < N){
        state.v = hmac(state.k, state.v)
        tmp = tmp || state.v
    }
    // Update state with no input
    state.k = hmac(state.k, state.v || 0x00)
    state.v = hmac(state.k, state.v)
    // Return the first N bits of tmp
    return tmp[0:N]
}
```
HMAC_DRBG Update

• Used instead of the "no-input update" when you have additional entropy on the generate call
• Used standalone for both instantiate (state.k = state.v = 0) and reseed (keep state.k and state.v)
• Designed so that even if the attacker controls the input but doesn't know k:
The attacker should not be able to predict the new k

```javascript
function hmac_drbg_update (state, input) {
    state.k = hmac(state.k, state.v || 0x00 || input)
    state.v = hmac(state.k, state.v)
    state.k = hmac(state.k, state.v || 0x01 || input)
    state.v = hmac(state.k, state.v)
}
```
Stream ciphers

- Block cipher: fixed-size, stateless, requires “modes” to securely process longer messages
- Stream cipher: keeps state from processing past message elements, can continually process new elements
- Common approach: “one-time pad on the cheap”:
  - XORs the plaintext with some “random” bits
- But: random bits ≠ the key (as in one-time pad)
  - Instead: output from cryptographically strong pseudorandom number generator (pRNG)
  - Anyone who actually calls this a "One Time Pad" is selling snake oil!
Building Stream Ciphers

- Encryption, given key $K$ and message $M$:
  - Choose a random value IV
  - $E(M, K) = pRNG(K, IV) \oplus M$

- Decryption, given key $K$, ciphertext $C$, and initialization vector IV:
  - $D(C, K) = PRNG(K, IV) \oplus C$

- Can encrypt message of any length because pRNG can produce any number of random bits...
  - But in practice, for an $n$-bit seed pRNG, stop at $2^{n/2}$. Because, of course...
Using a PRNG to Build a Stream Cipher

(Small) K, IV

PRNG

Keystream

M_i: i^{th} message of plaintext

Alice

Bob

(Small) K, IV

PRNG

Keystream

C_i

M_i
CTR mode is (mostly) a stream cipher

- $E(\text{ctr}, K)$ should look like a series of pseudo random numbers...
  - But after a large amount it is *slightly* distinguishable!
- Since it is actually a pseudo-random *permutation*...
  - For a cipher using 128b blocks, you will never get the same 128b number until you go all the way through the $2^{128}$ possible entries on the counter
  - Reason why you want to stop after $2^{64}$
    - if you are foolish enough to use CTR mode in the first place
- Also very minor information leakage:
  - If $C_i = C_j$, for $i \neq j$, it follows that $M_i \neq M_j$
UUID: Universally Unique Identifiers

- You got to have a "name" for something...
  - EG, to store a location in a filesystem
- Your name **must** be unique...
  - And your name **must** be unpredictable!
- Just chose a **random** value!
  - UUID: just chose a 128b random value
    - Well, it ends up being a 122b random value with some signaling information
    - A good UUID library uses a cryptographically-secure pRNG that is properly seeded
- Often written out in hex as:
  - 00112233-4455-6677-8899-aabbccddeeff
What Happens When The Random Numbers Goes Wrong...

- **Insufficient Entropy:**
  - Random number generator is seeded without enough entropy

- **Debian OpenSSL CVE-2008-0166**
  - In "cleaning up" OpenSSL (Debian 'bug' #363516), the author 'fixed' how OpenSSL seeds random numbers
    - Because the code, as written, caused Purify and Valgrind to complain about reading uninitialized memory
  - Unfortunate cleanup reduced the pRNG's seed to be **just** the process ID
    - So the pRNG would only start at one of ~30,000 starting points

- **This made it easy to find private keys**
  - Simply set to each possible starting point and generate a few private keys
  - See if you then find the corresponding public keys anywhere on the Internet

http://blog.dieweltistgarnichtso.net/Caprica,-2-years-ago
And Now Lets Add Some RNG Sabotage...

• The Dual_EC_DRBG
  • A pRNG pushed by the NSA behind the scenes based on Elliptic Curves
  • It relies on two parameters, $P$ and $Q$ on an elliptic curve
    • The person who generates $P$ and selects $Q=eP$ can predict the random number generator, regardless of the internal state
  • It also *sucked!*
    • It was horribly slow and even had subtle biases that shouldn't exist in a pRNG: You could distinguish the upper bits from random!

• Now this was spotted fairly early on...
  • Why should anyone use such a horrible random number generator?
Well, anyone not paid that is...

- RSA Data Security accepted 30 pieces of silver $10M from the NSA to implement Dual_EC in their RSA BSAFE library
  - And *silently* make it the default pRNG
- Using RSA's support, it became a NIST standard
  - And inserted into other products...
- And then the Snowden revelations
  - The initial discussion of this sabotage in the NY Times just vaguely referred to a Crypto talk given by Microsoft people...
    - That everybody quickly realized referred to Dual_EC
But this is insanely powerful...

- It isn't just forward prediction but being able to run the generator backwards!
  - Which is why Dual_EC is so nasty:
    Even if you know the internal state of HMAC_DRBG it has rollback resistance!
- In TLS (HTTPS) and Virtual Private Networks you have a motif of:
  - Generate a random session key
  - Generate some other random data that's public visible
    - EG, the IV in the encrypted channel, or the "random" nonce in TLS
    - Oh, and an NSA sponsored "standard" to spit out even more "random" bits!
- If you can run the random number generator backwards, you can find the session key
It Got Worse: Sabotaging Juniper

- Juniper also used Dual_EC in their Virtual Private Networks
  - "But we did it safely, we used a different Q"
- Sometime later, someone else noticed this...
  - "Hmm, P and Q are the keys to the backdoor...
    Lets just hack Juniper and rekey the lock!"
    - And whoever put in the first Dual_EC then went "Oh crap, we got locked out but we can’t do anything about it!"
- Sometime later, someone else goes...
  - "Hey, lets add an ssh backdoor"
- Sometime later, Juniper goes
  - "Whoops, someone added an ssh backdoor, lets see what else got F'ed with, oh, this # in the pRNG"
- And then everyone else went
  - "Ohh, patch for a backdoor. Lets see what got fixed.
    Oh, these look like Dual_EC parameters..."
Sabotaging "Magic Numbers"
In General

• Many cryptographic implementations depend on "magic" numbers
  • Parameters of an Elliptic curve
  • Magic points like $P$ and $Q$
  • Particular prime $p$ for Diffie/Hellman
  • The content of S-boxes in block cyphers

• Good systems should cleanly describe how they are generated
  • In some sound manner (e.g. AES's S-boxes)
  • In some "random" manner defined by a pRNG with a specific seed
    • Eg, seeded with "Nicholas Weaver Deserves Perfect Student Reviews"...
      Needs to be very low entropy so the designer can't try a gazillion seeds
Because Otherwise You Have Trouble...

- Not only Dual-EC's $P$ and $Q$
- Recent work: 1024b Diffie/Hellman moderately impractical...
  - But you can create a sabotaged prime that is 1/1,000,000 the work to crack!
    And the most often used "example" $p$'s origin is lost in time!
- It can cast doubt **even when a design is solid**:
  - The DES standard was developed by IBM but with input from the NSA
    - Everyone was suspicious about the NSA tampering with the S-boxes...
    - They did: The NSA made them **stronger** against an attack they knew but the public didn't
  - The NSA-defined elliptic curves P-256 and P-384
    - I trust them because they are in Suite-B/CNSA so the NSA uses them for TS communication:
      A backdoor here would be absolutely unacceptable...
      *but only because I actually believe the NSA wouldn't go and try to shoot itself in the head!*
So What To Use?

• AES-128-CFB or AES-256-CFB:
  • Robust to screwups encryption

• SHA-2 or SHA-3 family (256b, 384b, or 512b):
  • Robust cryptographic hashes, SHA-1 and MD5 are broken

• HMAC-SHA256 or HMAC-SHA3:
  • Different function than the encryption:
    Prevents screwups on using the same key & is a hash
  • Always Encrypt Then MAC!

• HMAC-SHA256-DRBG or HMAC-SHA3-DRBG:
  • The best pRNG available